Blockchain Lab
Experiment 4

Sanket More

D20A 35

Aim - Hands on Solidity Programming Assignments for creating Smart Contracts.

Theory -

Primitive Data Types, Variables, and Functions (Pure & View)

In Solidity, primitive data types are the basic building blocks of smart contracts. Commonly used data types include:

- **`uint`** / **`int`** – Unsigned and signed integers of various sizes (e.g., `uint256`, `int128`).
- **`bool`** – Represents logical values (`true` or `false`).
- **`address`** – Stores a 20-byte Ethereum address, typically used for user accounts or contract addresses.
- **`bytes`** / **`string`** – Used to store binary data or textual information.

Types of Variables

Solidity supports different categories of variables:

- **State variables** – Stored permanently on the blockchain.
- **Local variables** – Temporary variables created during function execution.
- **Global variables** – Predefined variables such as `msg.sender`, `msg.value`, and `block.timestamp`.

Functions in Solidity

Functions define the logic of a smart contract. Two important function types are:

- **`pure` functions** – Cannot read or modify blockchain state. They rely only on input parameters and internal calculations.
- **`view` functions** – Can read state variables but cannot modify them.

Using `pure` and `view` appropriately helps reduce gas costs and ensures function integrity.

Inputs and Outputs of Functions

Solidity functions can accept input parameters and return one or more output values.

- **Inputs** allow users or other contracts to pass data into a function.
- **Outputs** return results after performing computations.

For example, a function may accept an Ether amount and return whether a transaction was successful. Solidity also supports **named return variables**, which enhance code readability and simplify debugging.

Visibility, Modifiers, and Constructors

Function Visibility

Visibility determines who can access a function:

- **`public`** – Accessible from inside and outside the contract.
- **`private`** – Accessible only within the same contract.
- **`internal`** – Accessible within the contract and its derived (child) contracts.
- **`external`** – Can be called only from outside the contract.

Modifiers

Modifiers are reusable code blocks that alter function behavior. They are commonly used for access control, such as restricting certain functions to the contract owner (e.g., `onlyOwner`).

Constructors

A constructor is a special function that runs only once during contract deployment. It is typically used to initialize important variables, such as assigning the deployer as the contract owner.

Control Flow: if-else and Loops

Solidity's control flow mechanisms are similar to traditional programming languages:

- **`if-else` statements** enable conditional execution, such as verifying sufficient balance before transferring funds.
- **Loops (`for`, `while`, `do-while`)** allow repeated execution of code, such as iterating through an array.

However, loops must be used cautiously because excessive iterations increase gas consumption, making transactions more expensive.

Data Structures: Arrays, Mappings, Structs, and Enums

Solidity provides several data structures:

- **Arrays** – Store ordered lists of elements. They can be fixed-size or dynamic. Example: an array of user addresses.
- **Mappings** – Store key-value pairs for efficient lookups. Example:
  `mapping(address => uint)` for tracking account balances.
  Unlike arrays, mappings cannot be iterated directly.
- **Structs** – Custom data types that group related properties. Example:
  `struct Player { string name; uint score; }`
- **Enums** – Define a set of predefined constants, improving readability. Example:
  `enum Status { Pending, Active, Closed }`

Data Locations

Solidity uses three main data locations:

- **`storage`** – Permanently stored on the blockchain (used for state variables).
- **`memory`** – Temporary storage available only during function execution.
- **`calldata`** – Non-modifiable, non-persistent storage used for external function parameters. It is more gas-efficient than `memory`.

Understanding data locations is essential because they directly affect gas costs and contract performance.

Transactions: Ether, Wei, Gas, and Sending Transactions

Ether and Wei

Ether is the primary currency of Ethereum. All values are internally measured in **Wei**, the smallest unit:

1 Ether = 10^{18} Wei

Using Wei ensures high precision in financial calculations.

Gas and Gas Price

Every transaction consumes **gas**, which represents the computational effort required to execute it.

- **Gas price** determines how much Ether is paid per unit of gas.
- Higher gas prices encourage miners (validators) to prioritize a transaction.

Sending Transactions

Transactions are used to:

- Transfer Ether
- Interact with smart contracts

Common methods include:

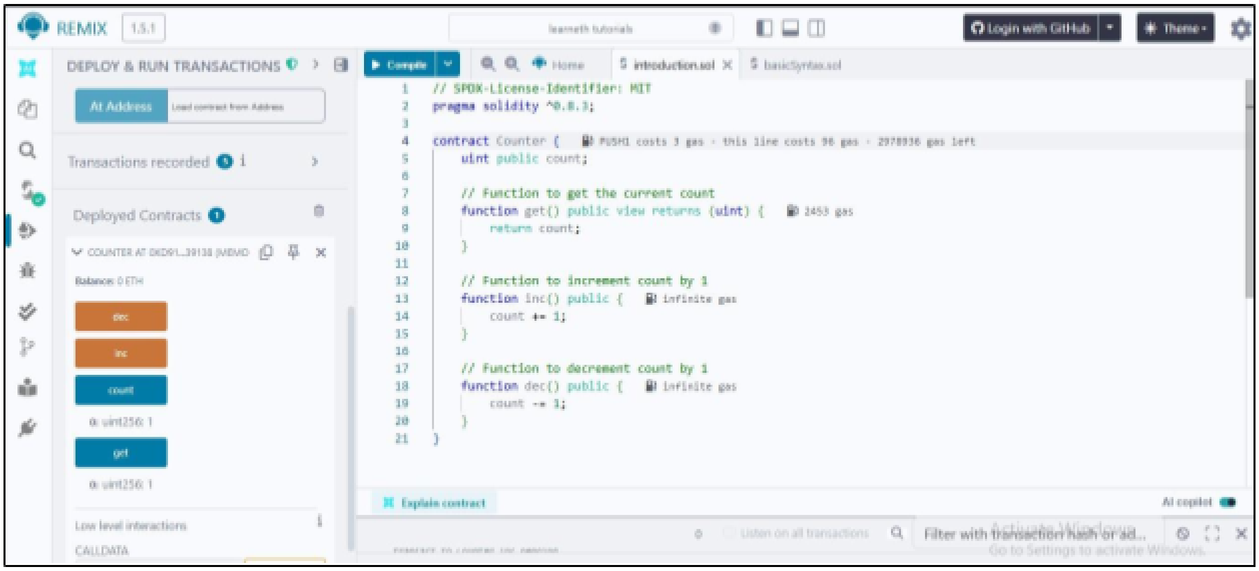
- `transfer()` – Sends Ether with fixed gas.
- `send()` – Similar to transfer but returns a boolean.
- `call()` – More flexible and commonly recommended for sending Ether.

Since every transaction requires gas, writing optimized and efficient smart contracts is crucial.

Code & Output -

Tutorial 1

1. Get counter value



```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.3;

contract Counter {    PUSH1 costs 3 gas - this line costs 98 gas - 2978936 gas left
    uint public count;

    // Function to get the current count
    function get() public view returns (uint) {    2453 gas
        return count;
    }

    // Function to increment count by 1
    function inc() public {    infinite gas
        count += 1;
    }

    // Function to decrement count by 1
    function dec() public {    infinite gas
        count -= 1;
    }
}
```

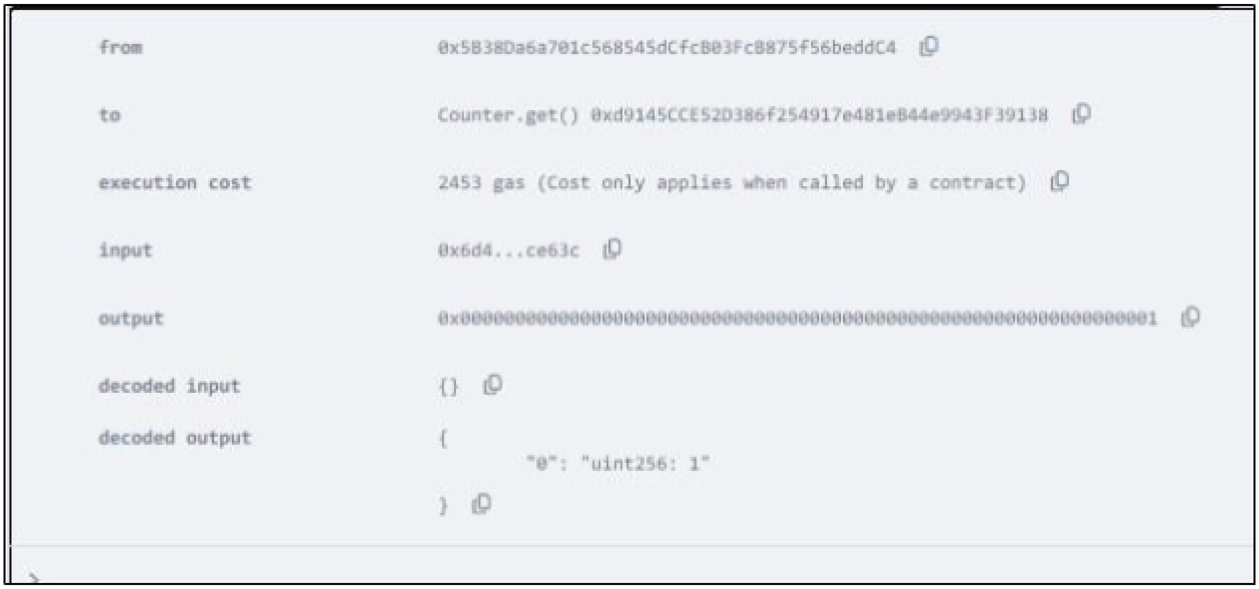
Balance: 0 ETH

count

0: uint256: 1

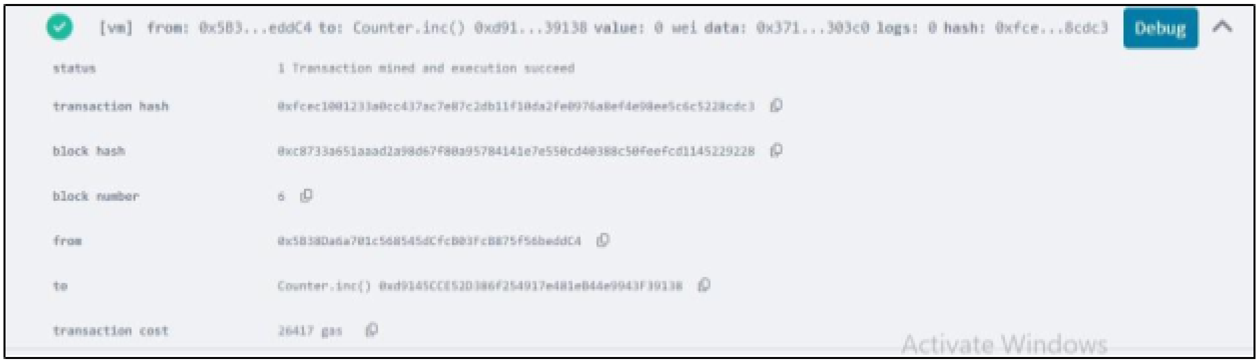
get

0: uint256: 1



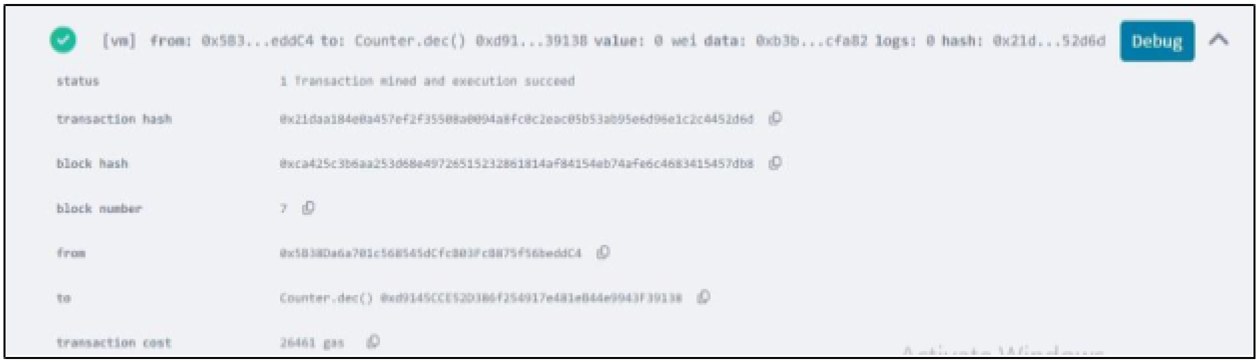
```
from                0x5B38Da6a701c568545dCfcB03FcB875f56beddC4

to                  Counter.get() 0xd9145CCE52D386f254917e481eB44e9943F39138

execution cost      2453 gas (Cost only applies when called by a contract)

input               0x6d4...ce63c

output              0x0000000000000000000000000000000000000000000000000000000000000001

decoded input       {}

decoded output      {
                        "0": "uint256: 1"
                    }
```

2. Increment counter value



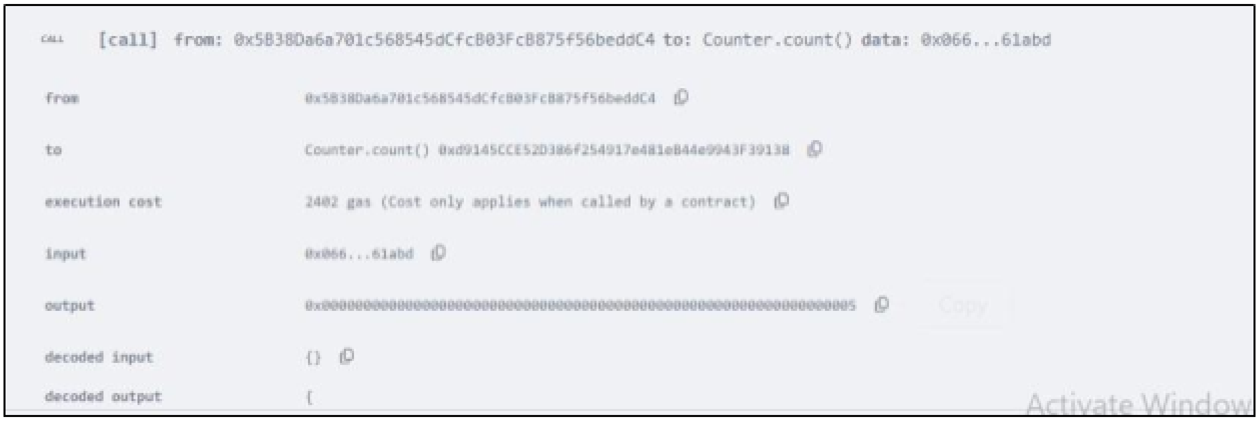
```
[vm] from: 0x5B3...eddC4 to: Counter.inc() 0xd91...39138 value: 0 wei data: 0x371...303c0 logs: 0 hash: 0xfce...8cdc3
status              1 Transaction mined and execution succeed

transaction hash    0xfcec1001233a0cc437ac7e87c2db11f10da2fe0976a8ef4e98ee5c6c5228cdc3

block hash          0xc8733a651aaad2a98d67f80a95784141e7e550cd40388c50feefcd1145229228

block number        6

from                0x5B38Da6a701c568545dCfcB03FcB875f56beddC4

to                  Counter.inc() 0xd9145CCE52D386f254917e481eB44e9943F39138

transaction cost    26417 gas
```

3. Decrement counter value



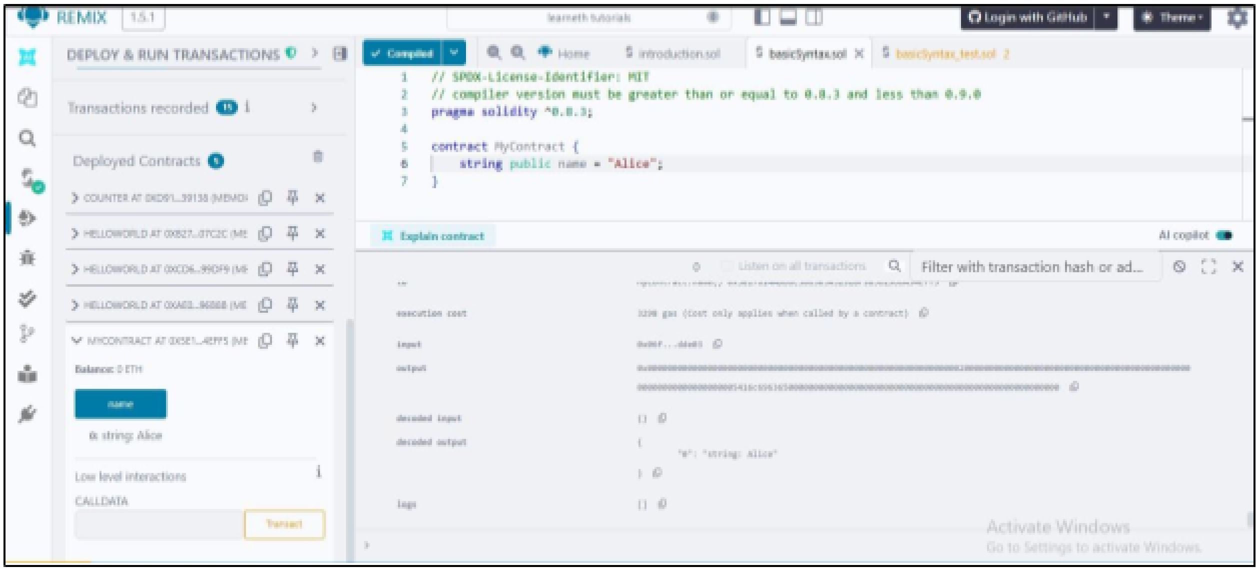
```
[vm] from: 0x5B3...eddC4 to: Counter.dec() 0xd91...39138 value: 0 wei data: 0xb3b...cfa82 logs: 0 hash: 0x21d...52d6d
status              1 Transaction mined and execution succeed

transaction hash    0x21daa184e0a457ef2f35508a0094a8fc0c2eac05b53ab95e6d96e1c2c4452d6d

block hash          0xca425c3b6aa253d68e4972651523286181 4af84154eb74afe6c4683415457db8

block number        7

from                0x5B38Da6a701c568545dCfcB03FcB875f56beddC4

to                  Counter.dec() 0xd9145CCE52D386f254917e481eB44e9943F39138

transaction cost    26461 gas
```

4. Get count value



```
CALL   [call] from: 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4 to: Counter.count() data: 0x066...61abd

from               0x5B38Da6a701c568545dCfcB03FcB875f56beddC4
to                 Counter.count() 0xd9145CCE52D386f254917e481eB44e9943F39138
execution cost     2402 gas (Cost only applies when called by a contract)
input              0x066...61abd
output             0x0000000000000000000000000000000000000000000000000000000000000005
decoded input      {}
decoded output     {
```

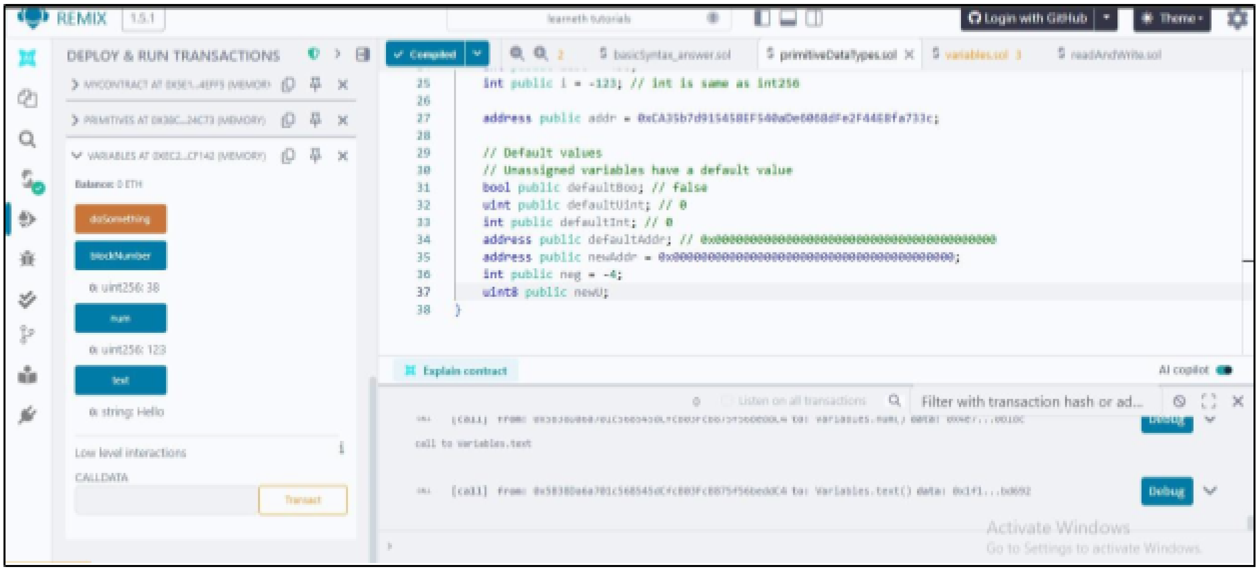
Tutorial 2



```
// SPDX-License-Identifier: MIT
// compiler version must be greater than or equal to 0.8.3 and less than 0.9.0
pragma solidity ^0.8.3;

contract MyContract {
    string public name = "Alice";
}
```

```
execution cost     3298 gas (Cost only applies when called by a contract)
input              0x06f...dde03
output             [illegible]
decoded input      {}
decoded output     {
                       "0": "string: Alice"
                   }
logs               []
```

Tutorial 3



Balance: 0 ETH

0: uint256: 38

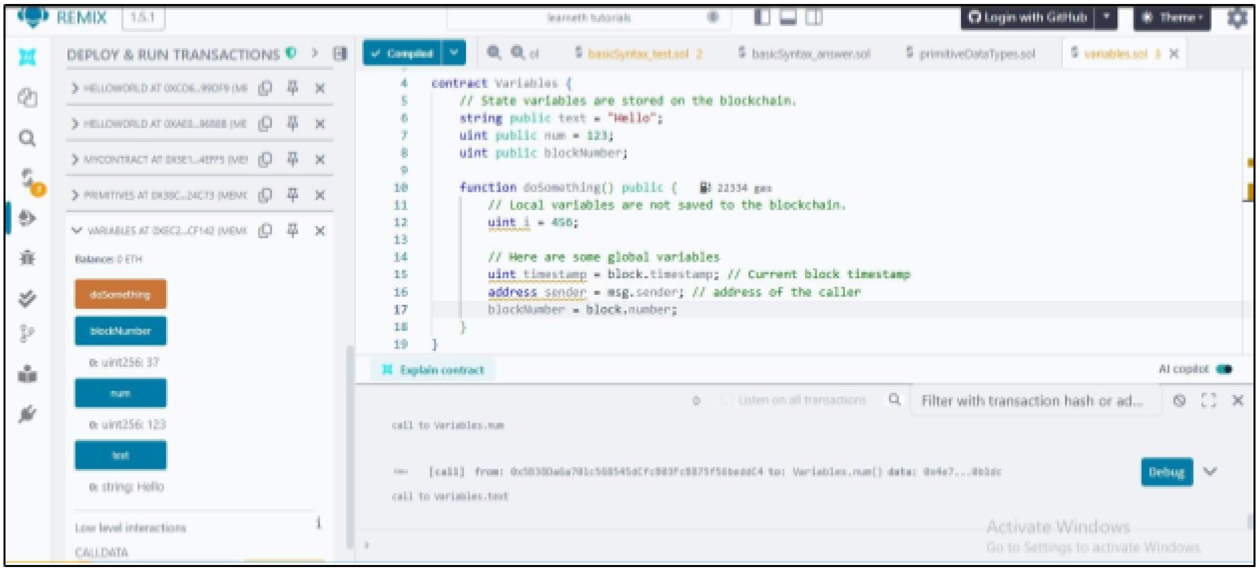
0: uint256: 123

0: string: Hello

```
    int public i = -123; // int is same as int256

    address public addr = 0xCA35b7d915458EF540aDe6068dFe2F44E8fa733c;

    // Default values
    // Unassigned variables have a default value
    bool public defaultBoo; // false
    uint public defaultUint; // 0
    int public defaultInt; // 0
    address public defaultAddr; // 0x0000000000000000000000000000000000000000
    address public newAddr = 0x0000000000000000000000000000000000000000;
    int public neg = -4;
    uint8 public newU;
}
```

```
call to Variables.text

[call] from: 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4 to: Variables.text() data: 0x1f1...bd692
```

Tutorial 4


REMIX
1.5.1
learneth tutorials
Login with GitHub
Theme
DEPLOY & RUN TRANSACTIONS
Compiled
basicSyntax_test.sol
basicSyntax_answer.sol
primitiveDataTypes.sol
variables.sol
HELLOWORLD AT 0XCD6...99DF9 (ME
HELLOWORLD AT 0XA03...96088 (ME
MYCONTRACT AT 0X5E1...4EFF5 (ME
PRIMITIVES AT 0X38C...24C73 (MEM
VARIABLES AT 0XEC2...CF142 (MEM
doSomething
blockNumber
num
text
Low level interactions
CALLDATA
Explain contract
AI copilot
Listen on all transactions
Filter with transaction hash or ad...
Debug

Balance: 0 ETH

0: uint256: 37

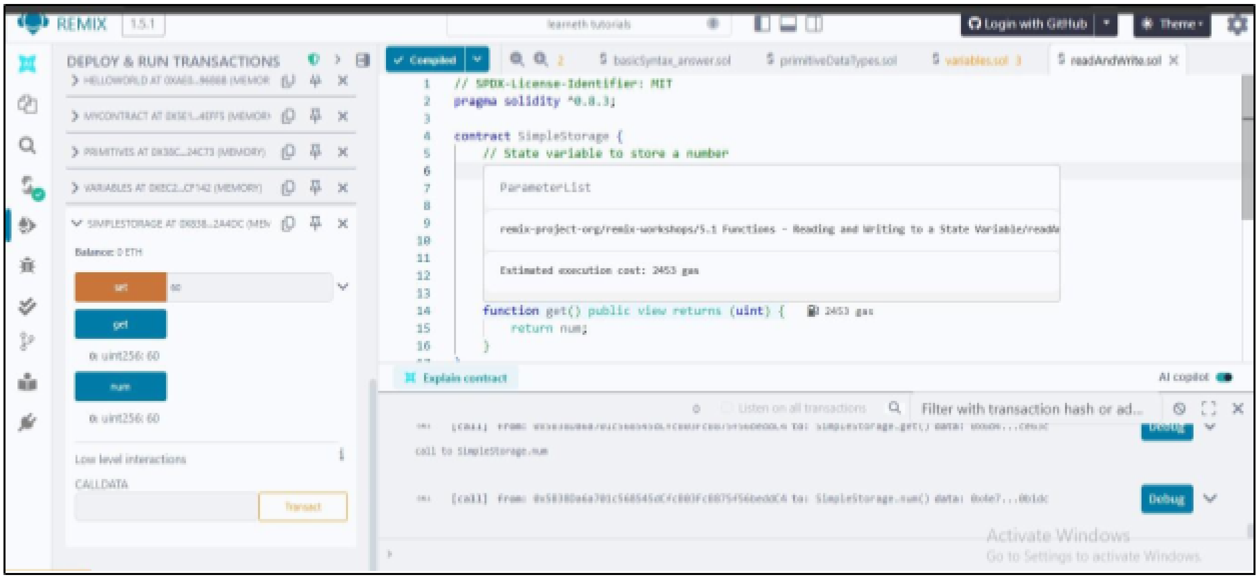
0: uint256: 123

0: string: Hello

```
contract Variables {
    // State variables are stored on the blockchain.
    string public text = "Hello";
    uint public num = 123;
    uint public blockNumber;

    function doSomething() public {    22334 gas
        // Local variables are not saved to the blockchain.
        uint i = 456;

        // Here are some global variables
        uint timestamp = block.timestamp; // Current block timestamp
        address sender = msg.sender; // address of the caller
        blockNumber = block.number;
    }
}
```

```
call to Variables.num

[call] from: 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4 to: Variables.num() data: 0x4e7...8b1dc
call to Variables.text
```

Tutorial 5


REMIX
1.5.1
learneth tutorials
Login with GitHub
Theme
DEPLOY & RUN TRANSACTIONS
HELLOWORLD AT 0XAE0...96888 (MEMOR
MYCONTRACT AT 0XSE1...4EFF5 (MEMOR
PRIMITIVES AT 0X30C...24C73 (MEMORY)
VARIABLES AT 0XEC2...CF142 (MEMORY)
SIMPLESTORAGE AT 0X838...2A4DC (MEM
Compiled
basicSyntax_answer.sol
primitiveDataTypes.sol
variables.sol
readAndWrite.sol
set
get
num
Low level interactions
CALLDATA
Transact
Explain contract
AI copilot
Listen on all transactions
Filter with transaction hash or ad...
Debug
Activate Windows
Go to Settings to activate Windows.

Balance: 0 ETH

set 60

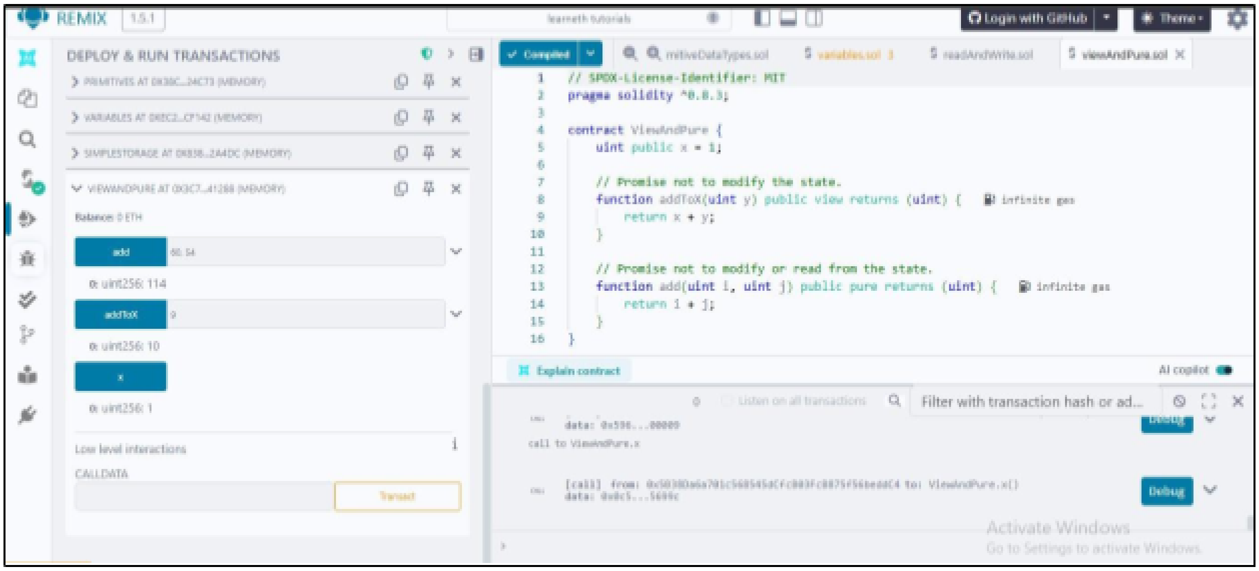
get — 0: uint256: 60

num — 0: uint256: 60

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.3;

contract SimpleStorage {
    // State variable to store a number

        ParameterList

        remix-project-org/remix-workshops/5.1 Functions - Reading and Writing to a State Variable/readW

        Estimated execution cost: 2453 gas

    function get() public view returns (uint) {    2453 gas
        return num;
    }
```

```
[call] from: 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4 to: SimpleStorage.get() data: 0x6d4...ce63c
call to SimpleStorage.num

[call] from: 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4 to: SimpleStorage.num() data: 0x4e7...0b1dc
```

Tutorial 6


REMIX
1.5.1
learneth tutorials
Login with GitHub
Theme
DEPLOY & RUN TRANSACTIONS
PRIMITIVES AT 0X30C...24C73 (MEMORY)
VARIABLES AT 0XEC2...CF142 (MEMORY)
SIMPLESTORAGE AT 0X838...2A4DC (MEMORY)
VIEWANDPURE AT 0X3C7...41288 (MEMORY)
Compiled
mitiveDataTypes.sol
variables.sol
readAndWrite.sol
viewAndPure.sol
add
addToX
x
Low level interactions
CALLDATA
Transact
Explain contract
AI copilot
Listen on all transactions
Filter with transaction hash or ad...
Debug
Activate Windows
Go to Settings to activate Windows.

Balance: 0 ETH

add 60, 54 — 0: uint256: 114

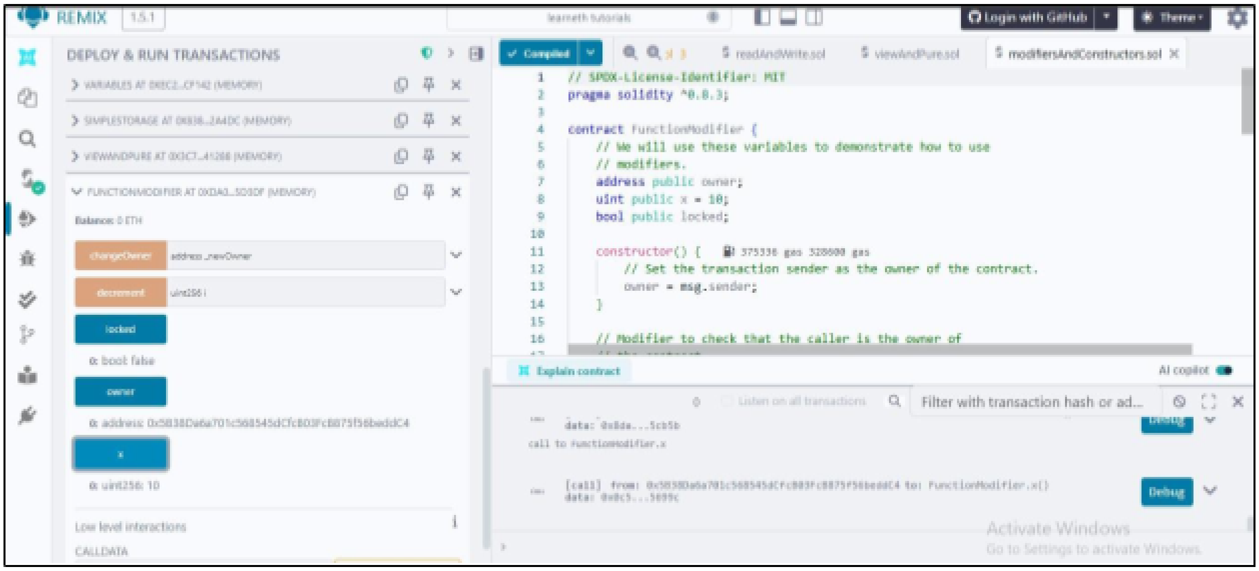
addToX 9 — 0: uint256: 10

x — 0: uint256: 1

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.3;

contract ViewAndPure {
    uint public x = 1;

    // Promise not to modify the state.
    function addToX(uint y) public view returns (uint) {    infinite gas
        return x + y;
    }

    // Promise not to modify or read from the state.
    function add(uint i, uint j) public pure returns (uint) {    infinite gas
        return i + j;
    }
}
```

```
data: 0x596...00009
call to ViewAndPure.x

[call] from: 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4 to: ViewAndPure.x()
data: 0x0c5...5699c
```

Tutorial 7


REMIX 1.5.1
learneth tutorials
Login with GitHub
Theme
DEPLOY & RUN TRANSACTIONS
VARIABLES AT 0XEC2...CF142 (MEMORY)
SIMPLESTORAGE AT 0X838...2A4DC (MEMORY)
VIEWANDPURE AT 0XC7...41288 (MEMORY)
FUNCTIONMODIFIER AT 0XDA0...5D3DF (MEMORY)
Compiled
readAndWrite.sol
viewAndPure.sol
modifiersAndConstructors.sol
Explain contract
AI copilot
Listen on all transactions
Filter with transaction hash or ad...
Debug
Activate Windows
Go to Settings to activate Windows.

Balance: 0 ETH

changeOwner address _newOwner

decrement uint256 i

locked

0: bool: false

owner

0: address: 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4

x

0: uint256: 10

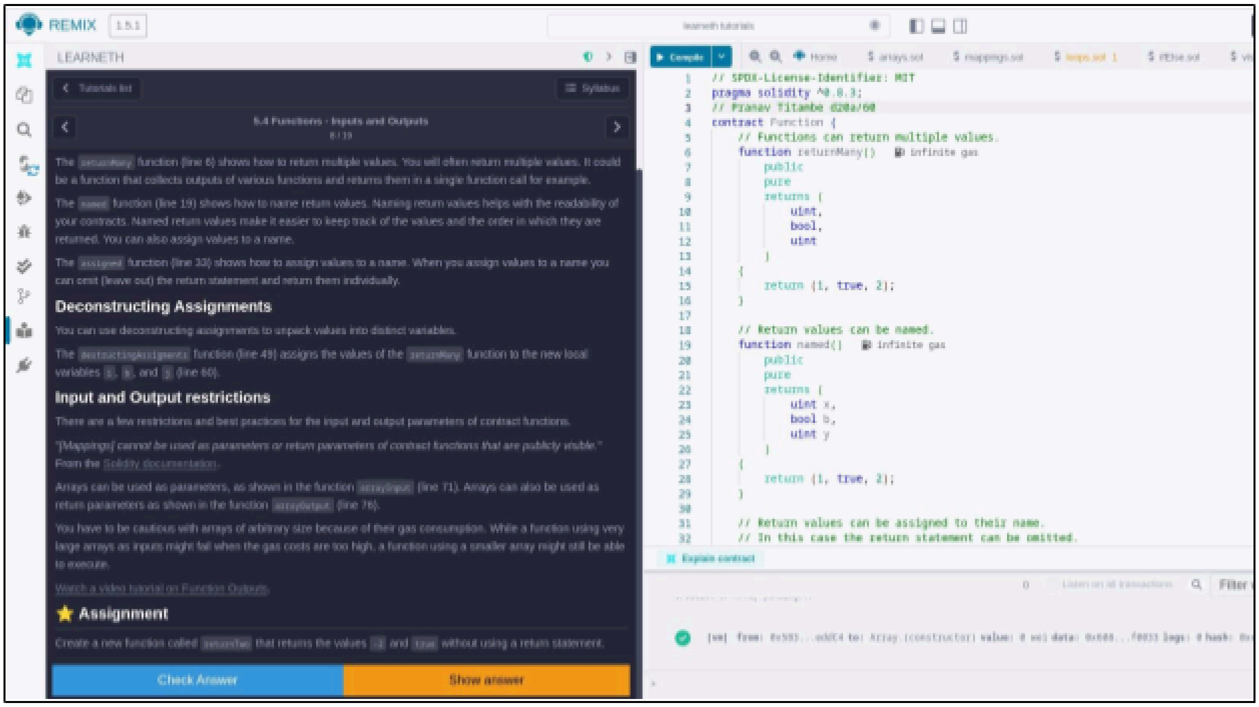
Low level interactions

CALLDATA

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.3;

contract FunctionModifier {
    // We will use these variables to demonstrate how to use
    // modifiers.
    address public owner;
    uint public x = 10;
    bool public locked;

    constructor() {     375336 gas 320600 gas
        // Set the transaction sender as the owner of the contract.
        owner = msg.sender;
    }

    // Modifier to check that the caller is the owner of
```

```
data: 0x8da...5cb5b
call to FunctionModifier.x

[call] from: 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4 to: FunctionModifier.x()
data: 0x0c5...5699c
```

Tutorial 8



5.4 Functions - Inputs and Outputs

8 / 19

The returnMany function (line 6) shows how to return multiple values. You will often return multiple values. It could be a function that collects outputs of various functions and returns them in a single function call for example.

The named function (line 19) shows how to name return values. Naming return values helps with the readability of your contracts. Named return values make it easier to keep track of the values and the order in which they are returned. You can also assign values to a name.

The assigned function (line 33) shows how to assign values to a name. When you assign values to a name you can omit (leave out) the return statement and return them individually.

Deconstructing Assignments

You can use deconstructing assignments to unpack values into distinct variables.

The destructingAssigments function (line 49) assigns the values of the returnMany function to the new local variables i, b, and j (line 60).

Input and Output restrictions

There are a few restrictions and best practices for the input and output parameters of contract functions.

"*[Mappings] cannot be used as parameters or return parameters of contract functions that are publicly visible.*" From the Solidity documentation.

Arrays can be used as parameters, as shown in the function arrayInput (line 71). Arrays can also be used as return parameters as shown in the function arrayOutput (line 76).

You have to be cautious with arrays of arbitrary size because of their gas consumption. While a function using very large arrays as inputs might fail when the gas costs are too high, a function using a smaller array might still be able to execute.

Watch a video tutorial on Function Outputs.

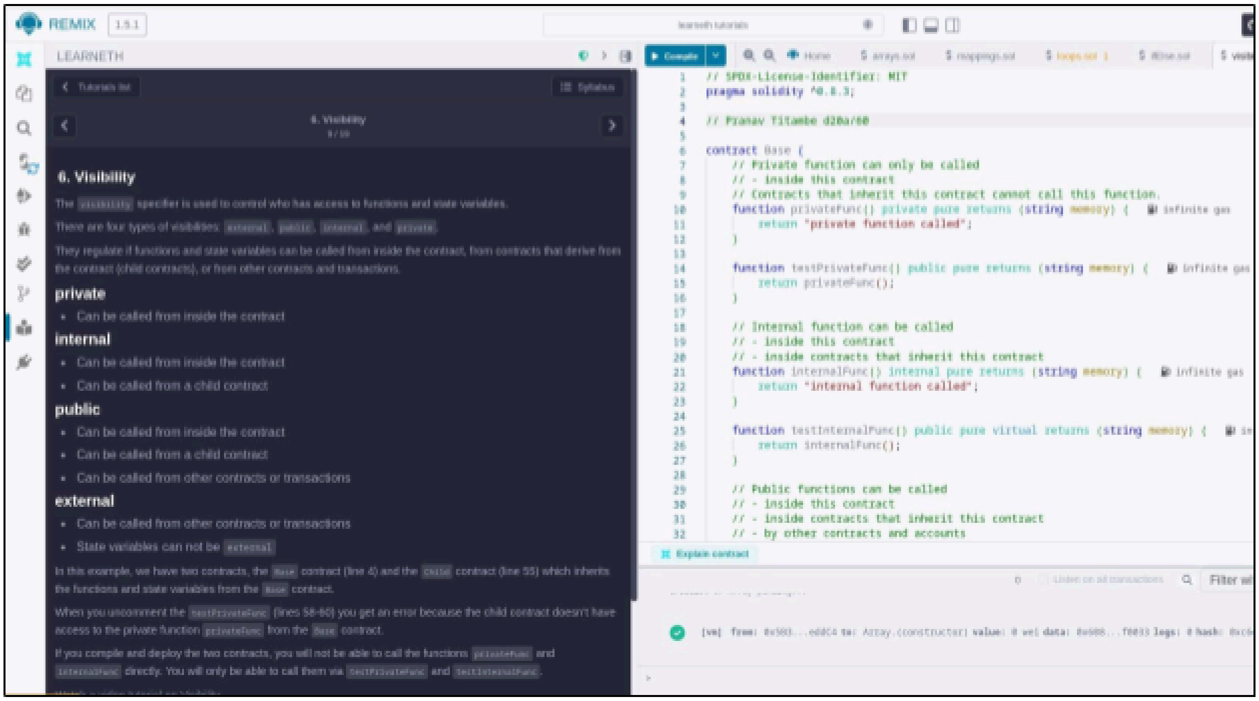
⭐ Assignment

Create a new function called returnTwo that returns the values -2 and true without using a return statement.

```
1   // SPDX-License-Identifier: MIT
2   pragma solidity ^0.8.3;
3   // Pranav Titambe d20a/60
4   contract Function {
5       // Functions can return multiple values.
6       function returnMany()     infinite gas
7           public
8           pure
9           returns (
10              uint,
11              bool,
12              uint
13          )
14      {
15          return (1, true, 2);
16      }
17
18      // Return values can be named.
19      function named()     infinite gas
20          public
21          pure
22          returns (
23              uint x,
24              bool b,
25              uint y
26          )
27      {
28          return (1, true, 2);
29      }
30
31      // Return values can be assigned to their name.
32      // In this case the return statement can be omitted.
```

```
[vm] from: 0x5B3...eddC4 to: Array.(constructor) value: 0 wei data: 0x608...f0033 logs: 0 hash: 0x
```

Tutorial 9



6. Visibility

The `visibility` specifier is used to control who has access to functions and state variables.

There are four types of visibilities: `external`, `public`, `internal`, and `private`.

They regulate if functions and state variables can be called from inside the contract, from contracts that derive from the contract (child contracts), or from other contracts and transactions.

private

- Can be called from inside the contract

internal

- Can be called from inside the contract
- Can be called from a child contract

public

- Can be called from inside the contract
- Can be called from a child contract
- Can be called from other contracts or transactions

external

- Can be called from other contracts or transactions
- State variables can not be `external`

In this example, we have two contracts, the `Base` contract (line 4) and the `Child` contract (line 55) which inherits the functions and state variables from the `Base` contract.

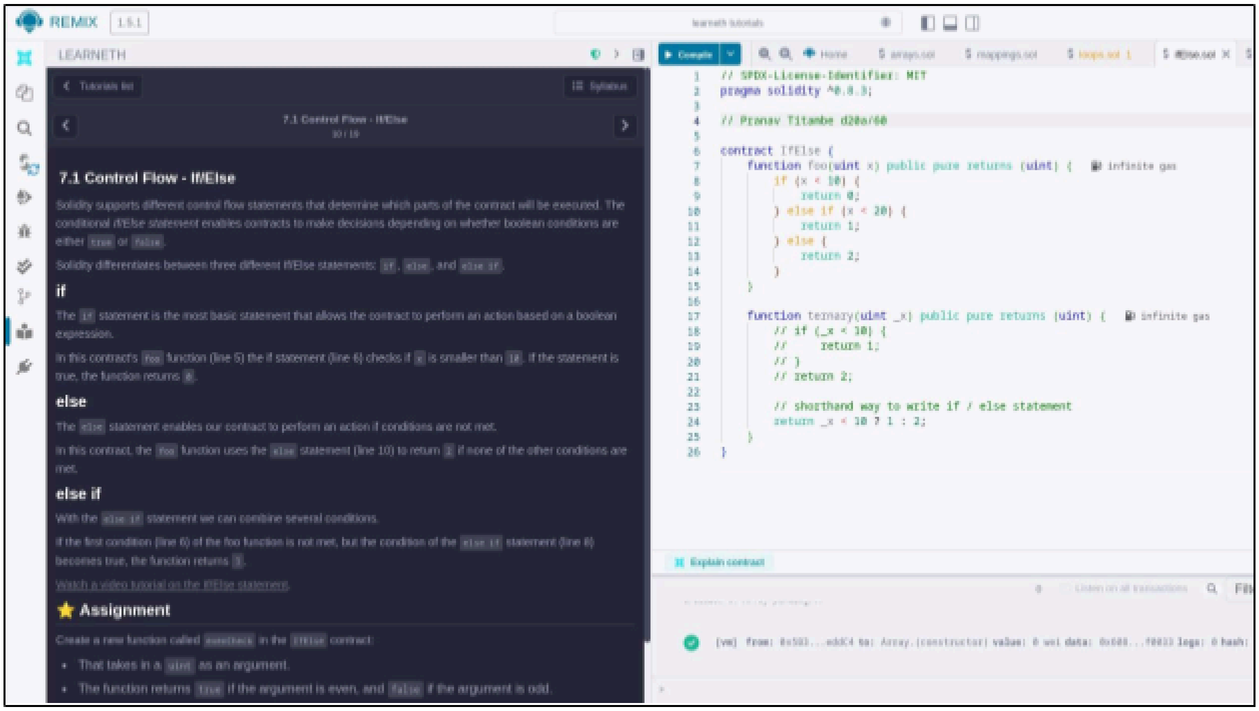
When you uncomment the `testPrivateFunc` (lines 58-60) you get an error because the child contract doesn't have access to the private function `privateFunc` from the `Base` contract.

If you compile and deploy the two contracts, you will not be able to call the functions `privateFunc` and `internalFunc` directly. You will only be able to call them via `testPrivateFunc` and `testInternalFunc`.

```
1   // SPDX-License-Identifier: MIT
2   pragma solidity ^0.8.3;
3
4   // Pranav Titambe d20a/60
5
6   contract Base {
7       // Private function can only be called
8       // - inside this contract
9       // Contracts that inherit this contract cannot call this function.
10      function privateFunc() private pure returns (string memory) {    infinite gas
11          return "private function called";
12      }
13
14      function testPrivateFunc() public pure returns (string memory) {    infinite gas
15          return privateFunc();
16      }
17
18      // Internal function can be called
19      // - inside this contract
20      // - inside contracts that inherit this contract
21      function internalFunc() internal pure returns (string memory) {    infinite gas
22          return "internal function called";
23      }
24
25      function testInternalFunc() public pure virtual returns (string memory) {    in
26          return internalFunc();
27      }
28
29      // Public functions can be called
30      // - inside this contract
31      // - inside contracts that inherit this contract
32      // - by other contracts and accounts
```

```
[vm] from: 0x5B3...eddC4 to: Array.(constructor) value: 0 wei data: 0x608...f0033 logs: 0 hash: 0xc6
```

Tutorial 10



7.1 Control Flow - If/Else

Solidity supports different control flow statements that determine which parts of the contract will be executed. The conditional If/Else statement enables contracts to make decisions depending on whether boolean conditions are either `true` or `false`.

Solidity differentiates between three different If/Else statements: `if`, `else`, and `else if`.

if

The `if` statement is the most basic statement that allows the contract to perform an action based on a boolean expression.

In this contract's `foo` function (line 5) the if statement (line 6) checks if `x` is smaller than `10`. If the statement is true, the function returns `0`.

else

The `else` statement enables our contract to perform an action if conditions are not met.

In this contract, the `foo` function uses the `else` statement (line 10) to return `2` if none of the other conditions are met.

else if

With the `else if` statement we can combine several conditions.

If the first condition (line 6) of the foo function is not met, but the condition of the `else if` statement (line 8) becomes true, the function returns `1`.

Watch a video tutorial on the If/Else statement.

⭐ Assignment

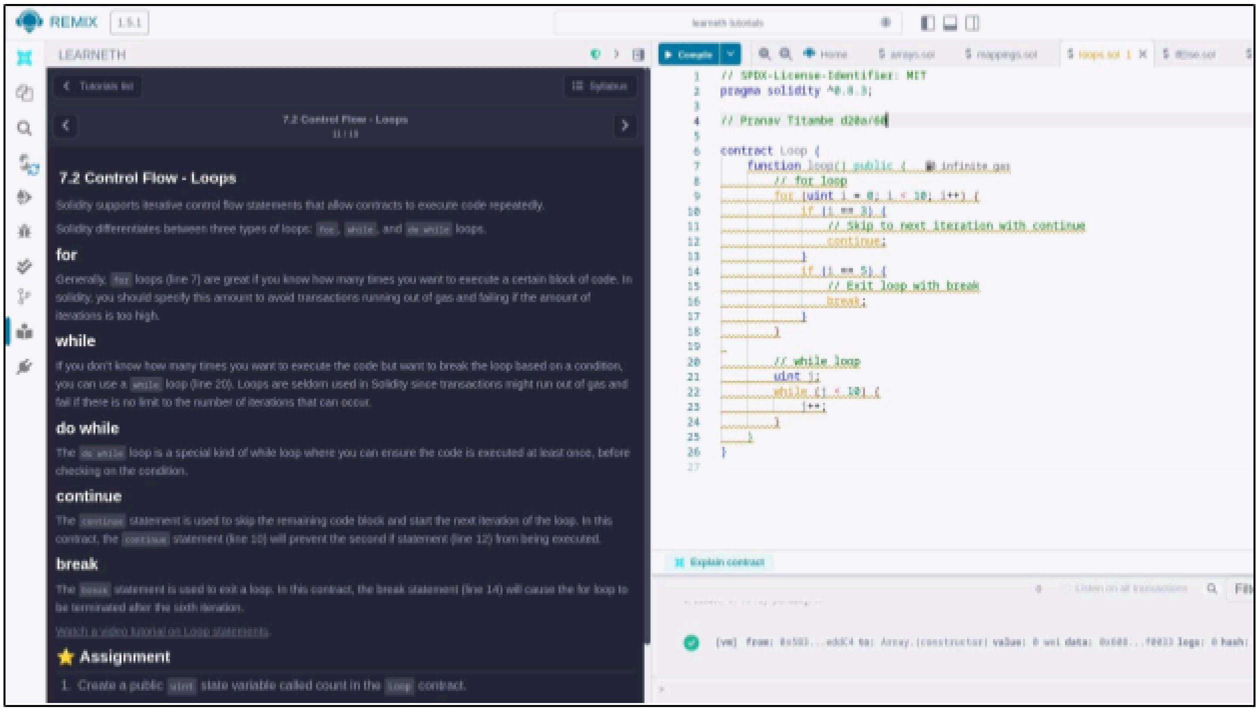
Create a new function called `evenCheck` in the `IfElse` contract:

- That takes in a `uint` as an argument.
- The function returns `true` if the argument is even, and `false` if the argument is odd.

```
1   // SPDX-License-Identifier: MIT
2   pragma solidity ^0.8.3;
3
4   // Pranav Titambe d20a/60
5
6   contract IfElse {
7       function foo(uint x) public pure returns (uint) {    infinite gas
8           if (x < 10) {
9               return 0;
10          } else if (x < 20) {
11              return 1;
12          } else {
13              return 2;
14          }
15      }
16
17      function ternary(uint _x) public pure returns (uint) {    infinite gas
18          // if (_x < 10) {
19          //     return 1;
20          // }
21          // return 2;
22
23          // shorthand way to write if / else statement
24          return _x < 10 ? 1 : 2;
25      }
26  }
```

```
[vm] from: 0x5B3...eddC4 to: Array.(constructor) value: 0 wei data: 0x608...f0033 logs: 0 hash:
```

Tutorial 11



7.2 Control Flow - Loops
11 / 13

7.2 Control Flow - Loops

Solidity supports iterative control flow statements that allow contracts to execute code repeatedly.

Solidity differentiates between three types of loops: `for`, `while`, and `do while` loops.

for

Generally, `for` loops (line 7) are great if you know how many times you want to execute a certain block of code. In solidity, you should specify this amount to avoid transactions running out of gas and failing if the amount of iterations is too high.

while

If you don't know how many times you want to execute the code but want to break the loop based on a condition, you can use a `while` loop (line 20). Loops are seldom used in Solidity since transactions might run out of gas and fail if there is no limit to the number of iterations that can occur.

do while

The `do while` loop is a special kind of while loop where you can ensure the code is executed at least once, before checking on the condition.

continue

The `continue` statement is used to skip the remaining code block and start the next iteration of the loop. In this contract, the `continue` statement (line 10) will prevent the second if statement (line 12) from being executed.

break

The `break` statement is used to exit a loop. In this contract, the break statement (line 14) will cause the for loop to be terminated after the sixth iteration.

Watch a video tutorial on Loop statements.

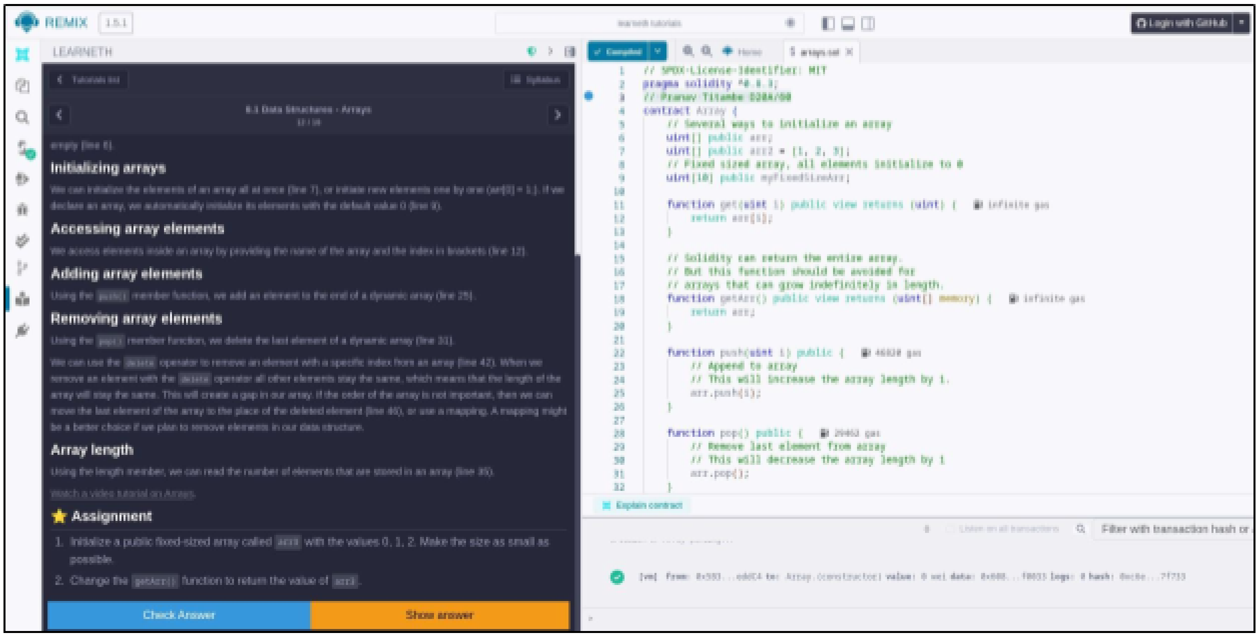
⭐ Assignment

1. Create a public `uint` state variable called count in the `Loop` contract.

```
1   // SPDX-License-Identifier: MIT
2   pragma solidity ^0.8.3;
3
4   // Pranav Titambe d20a/60
5
6   contract Loop {
7       function loop() public {    infinite gas
8           // for loop
9           for (uint i = 0; i < 10; i++) {
10              if (i == 3) {
11                  // Skip to next iteration with continue
12                  continue;
13              }
14              if (i == 5) {
15                  // Exit loop with break
16                  break;
17              }
18          }
19
20          // while loop
21          uint j;
22          while (j < 10) {
23              j++;
24          }
25      }
26  }
27
```

```
[vm] from: 0x5B3...eddC4 to: Array.(constructor) value: 0 wei data: 0x608...f0033 logs: 0 hash:
```

Tutorial 12



8.1 Data Structures - Arrays
12 / 19

empty (line 6).

Initializing arrays

We can initialize the elements of an array all at once (line 7), or initiate new elements one by one (arr[0] = 1;). If we declare an array, we automatically initialize its elements with the default value 0 (line 9).

Accessing array elements

We access elements inside an array by providing the name of the array and the index in brackets (line 12).

Adding array elements

Using the `push()` member function, we add an element to the end of a dynamic array (line 25).

Removing array elements

Using the `pop()` member function, we delete the last element of a dynamic array (line 31).

We can use the `delete` operator to remove an element with a specific index from an array (line 42). When we remove an element with the `delete` operator all other elements stay the same, which means that the length of the array will stay the same. This will create a gap in our array. If the order of the array is not important, then we can move the last element of the array to the place of the deleted element (line 46), or use a mapping. A mapping might be a better choice if we plan to remove elements in our data structure.

Array length

Using the length member, we can read the number of elements that are stored in an array (line 35).

Watch a video tutorial on Arrays.

⭐ Assignment

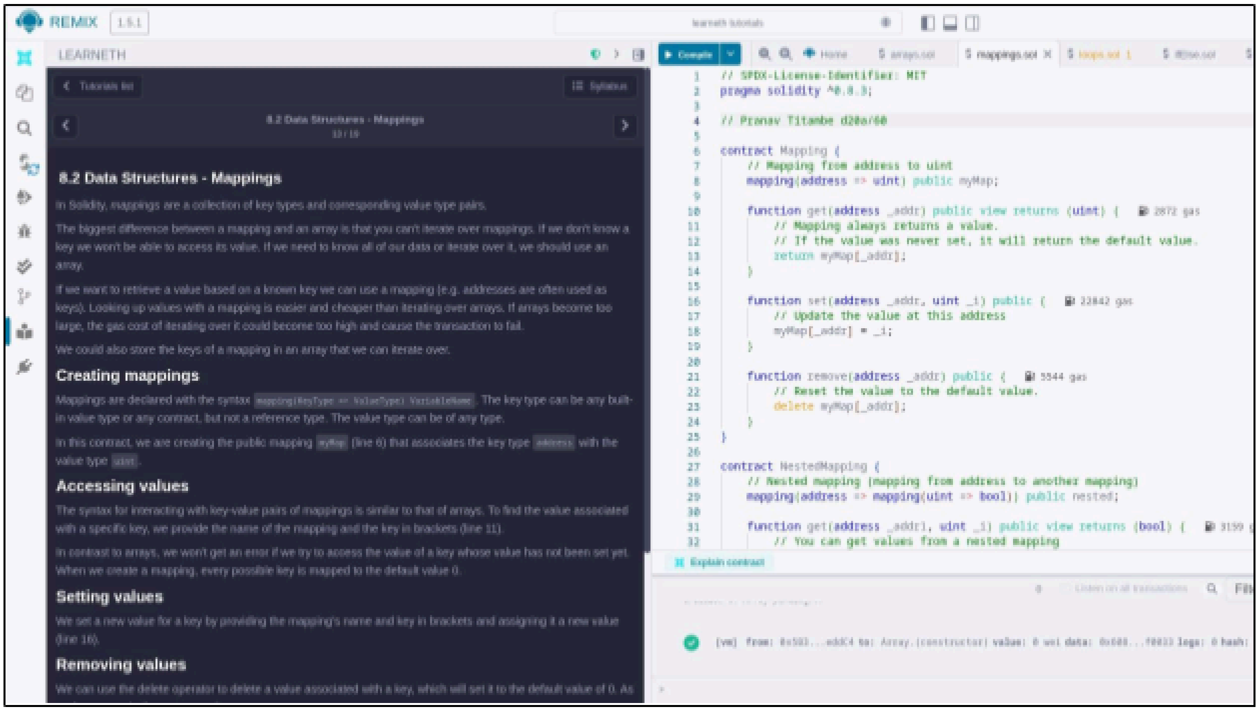
1. Initialize a public fixed-sized array called `arr3` with the values 0, 1, 2. Make the size as small as possible.
2. Change the `getArr()` function to return the value of `arr3`.

Check Answer | Show answer

```
1   // SPDX-License-Identifier: MIT
2   pragma solidity ^0.8.3;
3   // Pranav Titambe D20A/60
4   contract Array {
5       // Several ways to initialize an array
6       uint[] public arr;
7       uint[] public arr2 = [1, 2, 3];
8       // Fixed sized array, all elements initialize to 0
9       uint[10] public myFixedSizeArr;
10
11      function get(uint i) public view returns (uint) {    infinite gas
12          return arr[i];
13      }
14
15      // Solidity can return the entire array.
16      // But this function should be avoided for
17      // arrays that can grow indefinitely in length.
18      function getArr() public view returns (uint[] memory) {    infinite gas
19          return arr;
20      }
21
22      function push(uint i) public {    46820 gas
23          // Append to array
24          // This will increase the array length by 1.
25          arr.push(i);
26      }
27
28      function pop() public {    29462 gas
29          // Remove last element from array
30          // This will decrease the array length by 1
31          arr.pop();
32      }
```

```
[vm] from: 0x5B3...eddC4 to: Array.(constructor) value: 0 wei data: 0x608...f0033 logs: 0 hash: 0xc6e...7f733
```

Tutorial 13



8.2 Data Structures - Mappings

In Solidity, mappings are a collection of key types and corresponding value type pairs.

The biggest difference between a mapping and an array is that you can't iterate over mappings. If we don't know a key we won't be able to access its value. If we need to know all of our data or iterate over it, we should use an array.

If we want to retrieve a value based on a known key we can use a mapping (e.g. addresses are often used as keys). Looking up values with a mapping is easier and cheaper than iterating over arrays. If arrays become too large, the gas cost of iterating over it could become too high and cause the transaction to fail.

We could also store the keys of a mapping in an array that we can iterate over.

Creating mappings

Mappings are declared with the syntax `mapping(KeyType => ValueType) VariableName`. The key type can be any built-in value type or any contract, but not a reference type. The value type can be of any type.

In this contract, we are creating the public mapping `myMap` (line 6) that associates the key type `address` with the value type `uint`.

Accessing values

The syntax for interacting with key-value pairs of mappings is similar to that of arrays. To find the value associated with a specific key, we provide the name of the mapping and the key in brackets (line 11).

In contrast to arrays, we won't get an error if we try to access the value of a key whose value has not been set yet. When we create a mapping, every possible key is mapped to the default value 0.

Setting values

We set a new value for a key by providing the mapping's name and key in brackets and assigning it a new value (line 16).

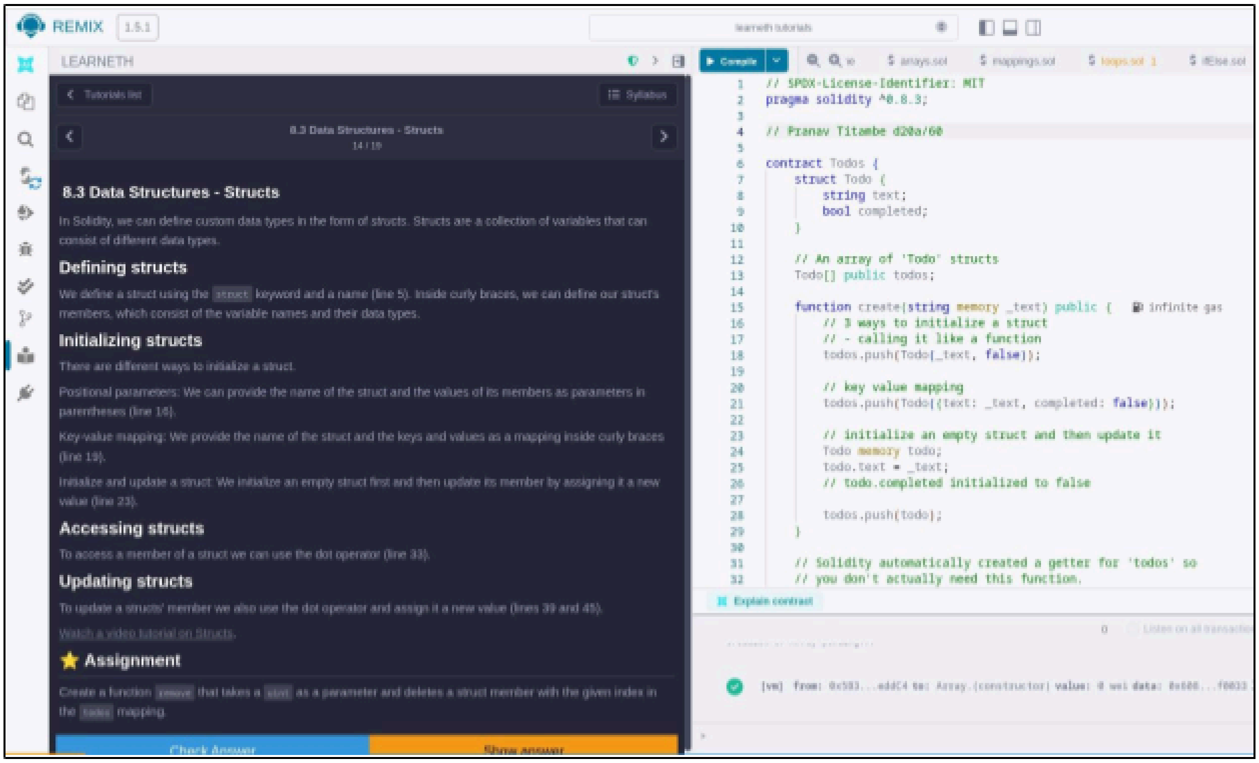
Removing values

We can use the delete operator to delete a value associated with a key, which will set it to the default value of 0. As

```
1   // SPDX-License-Identifier: MIT
2   pragma solidity ^0.8.3;
3
4   // Pranav Titambe d20a/60
5
6   contract Mapping {
7       // Mapping from address to uint
8       mapping(address => uint) public myMap;
9
10      function get(address _addr) public view returns (uint) {    2872 gas
11          // Mapping always returns a value.
12          // If the value was never set, it will return the default value.
13          return myMap[_addr];
14      }
15
16      function set(address _addr, uint _i) public {    22842 gas
17          // Update the value at this address
18          myMap[_addr] = _i;
19      }
20
21      function remove(address _addr) public {    5544 gas
22          // Reset the value to the default value.
23          delete myMap[_addr];
24      }
25  }
26
27  contract NestedMapping {
28      // Nested mapping (mapping from address to another mapping)
29      mapping(address => mapping(uint => bool)) public nested;
30
31      function get(address _addr1, uint _i) public view returns (bool) {    3159 g
32          // You can get values from a nested mapping
```

```
[vm] from: 0x5B3...eddC4 to: Array.(constructor) value: 0 wei data: 0x608...f0033 logs: 0 hash:
```

Tutorial 14



8.3 Data Structures - Structs

In Solidity, we can define custom data types in the form of structs. Structs are a collection of variables that can consist of different data types.

Defining structs

We define a struct using the `struct` keyword and a name (line 5). Inside curly braces, we can define our struct's members, which consist of the variable names and their data types.

Initializing structs

There are different ways to initialize a struct.

Positional parameters: We can provide the name of the struct and the values of its members as parameters in parentheses (line 16).

Key-value mapping: We provide the name of the struct and the keys and values as a mapping inside curly braces (line 19).

Initialize and update a struct: We initialize an empty struct first and then update its member by assigning it a new value (line 23).

Accessing structs

To access a member of a struct we can use the dot operator (line 33).

Updating structs

To update a structs' member we also use the dot operator and assign it a new value (lines 39 and 45).

Watch a video tutorial on Structs.

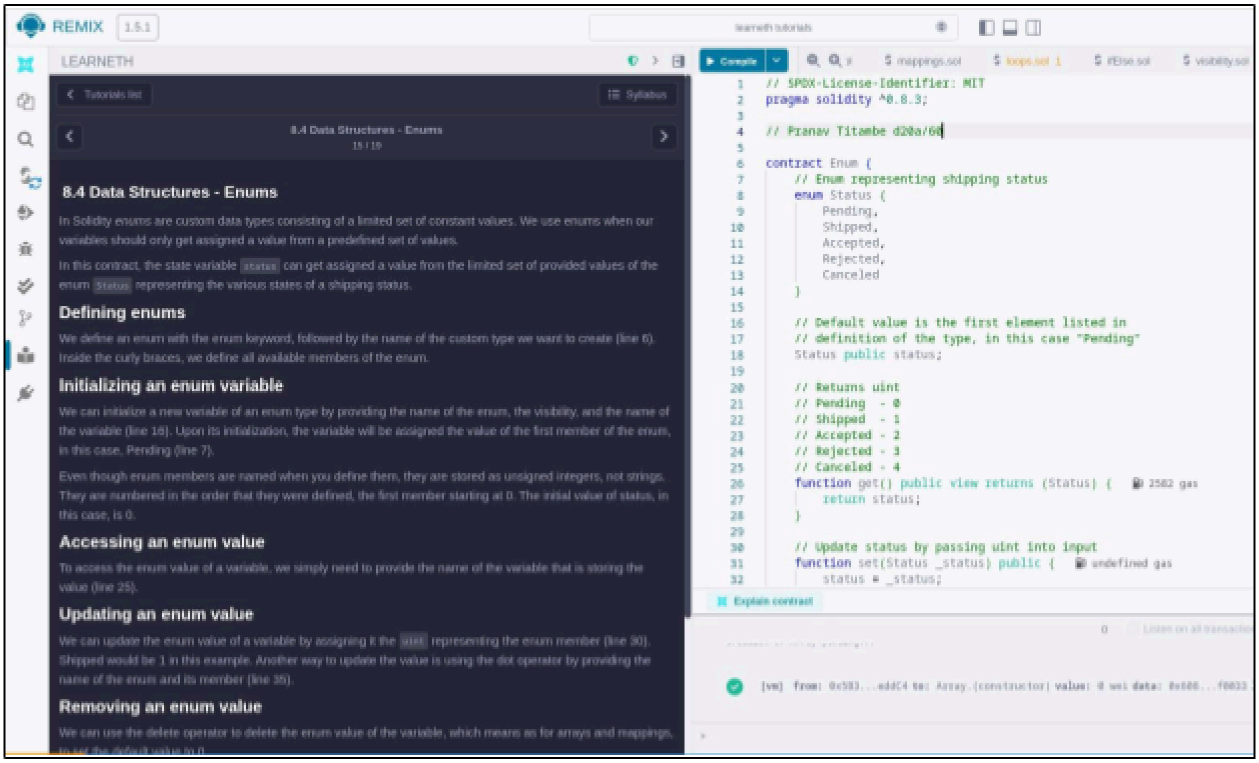
⭐ Assignment

Create a function `remove` that takes a `uint` as a parameter and deletes a struct member with the given index in the `todos` mapping.

```
1   // SPDX-License-Identifier: MIT
2   pragma solidity ^0.8.3;
3
4   // Pranav Titambe d20a/60
5
6   contract Todos {
7       struct Todo {
8           string text;
9           bool completed;
10      }
11
12      // An array of 'Todo' structs
13      Todo[] public todos;
14
15      function create(string memory _text) public {    infinite gas
16          // 3 ways to initialize a struct
17          // - calling it like a function
18          todos.push(Todo(_text, false));
19
20          // key value mapping
21          todos.push(Todo({text: _text, completed: false}));
22
23          // initialize an empty struct and then update it
24          Todo memory todo;
25          todo.text = _text;
26          // todo.completed initialized to false
27
28          todos.push(todo);
29      }
30
31      // Solidity automatically created a getter for 'todos' so
32      // you don't actually need this function.
```

```
[vm] from: 0x5B3...eddC4 to: Array.(constructor) value: 0 wei data: 0x608...f0033
```

Tutorial 15



8.4 Data Structures - Enums
15 / 19

8.4 Data Structures - Enums

In Solidity enums are custom data types consisting of a limited set of constant values. We use enums when our variables should only get assigned a value from a predefined set of values.

In this contract, the state variable `status` can get assigned a value from the limited set of provided values of the enum `Status` representing the various states of a shipping status.

Defining enums

We define an enum with the enum keyword, followed by the name of the custom type we want to create (line 6). Inside the curly braces, we define all available members of the enum.

Initializing an enum variable

We can initialize a new variable of an enum type by providing the name of the enum, the visibility, and the name of the variable (line 16). Upon its initialization, the variable will be assigned the value of the first member of the enum, in this case, Pending (line 7).

Even though enum members are named when you define them, they are stored as unsigned integers, not strings. They are numbered in the order that they were defined, the first member starting at 0. The initial value of status, in this case, is 0.

Accessing an enum value

To access the enum value of a variable, we simply need to provide the name of the variable that is storing the value (line 25).

Updating an enum value

We can update the enum value of a variable by assigning it the `uint` representing the enum member (line 30). Shipped would be 1 in this example. Another way to update the value is using the dot operator by providing the name of the enum and its member (line 35).

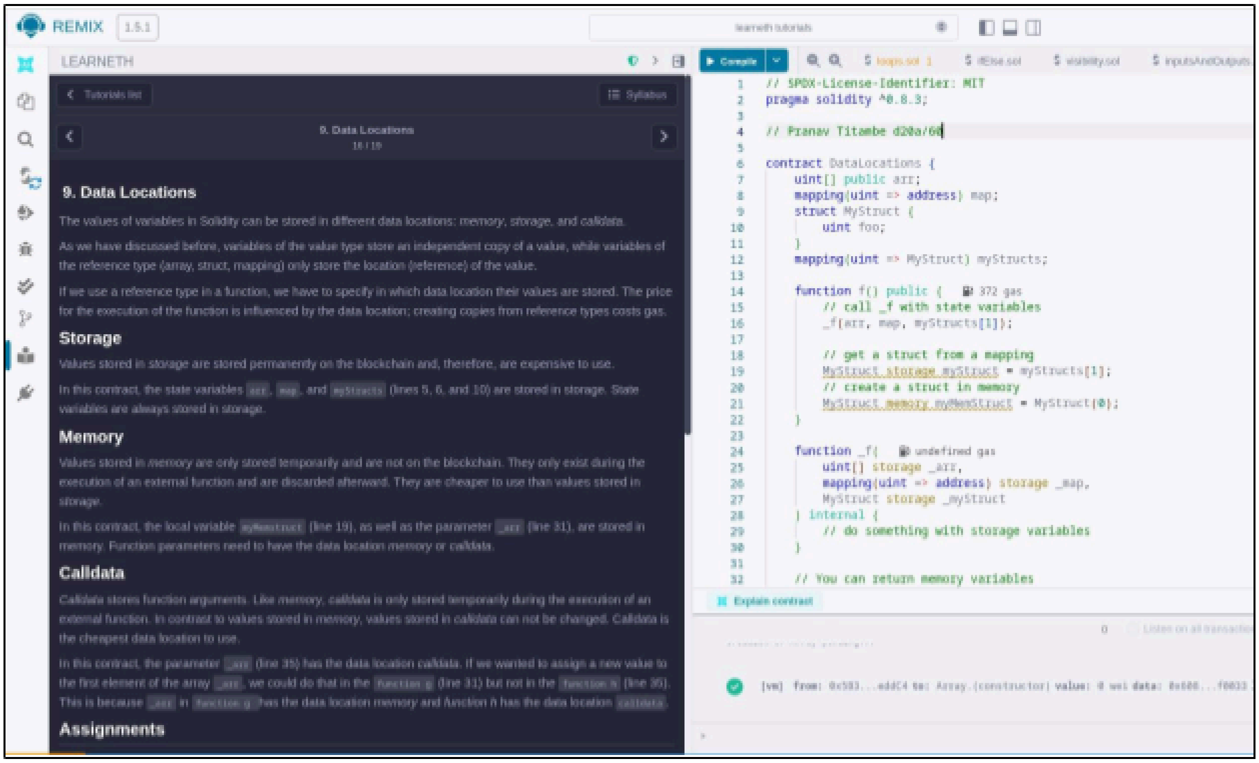
Removing an enum value

We can use the delete operator to delete the enum value of the variable, which means as for arrays and mappings, to set the default value to 0.

```
1   // SPDX-License-Identifier: MIT
2   pragma solidity ^0.8.3;
3
4   // Pranav Titambe d20a/60
5
6   contract Enum {
7       // Enum representing shipping status
8       enum Status {
9           Pending,
10          Shipped,
11          Accepted,
12          Rejected,
13          Canceled
14      }
15
16      // Default value is the first element listed in
17      // definition of the type, in this case "Pending"
18      Status public status;
19
20      // Returns uint
21      // Pending  - 0
22      // Shipped  - 1
23      // Accepted - 2
24      // Rejected - 3
25      // Canceled - 4
26      function get() public view returns (Status) {    2582 gas
27          return status;
28      }
29
30      // Update status by passing uint into input
31      function set(Status _status) public {    undefined gas
32          status = _status;
```

```
[vm] from: 0x5B3...eddC4 to: Array.(constructor) value: 0 wei data: 0x608...f0033
```

Tutorial 16



9. Data Locations
16 / 19

9. Data Locations

The values of variables in Solidity can be stored in different data locations: memory, storage, and calldata.

As we have discussed before, variables of the value type store an independent copy of a value, while variables of the reference type (array, struct, mapping) only store the location (reference) of the value.

If we use a reference type in a function, we have to specify in which data location their values are stored. The price for the execution of the function is influenced by the data location; creating copies from reference types costs gas.

Storage

Values stored in storage are stored permanently on the blockchain and, therefore, are expensive to use.

In this contract, the state variables `arr`, `map`, and `myStructs` (lines 5, 6, and 10) are stored in storage. State variables are always stored in storage.

Memory

Values stored in *memory* are only stored temporarily and are not on the blockchain. They only exist during the execution of an external function and are discarded afterward. They are cheaper to use than values stored in storage.

In this contract, the local variable `myMemstruct` (line 19), as well as the parameter `_arr` (line 31), are stored in memory. Function parameters need to have the data location memory or calldata.

Calldata

Calldata stores function arguments. Like memory, calldata is only stored temporarily during the execution of an external function. In contrast to values stored in memory, values stored in calldata can not be changed. Calldata is the cheapest data location to use.

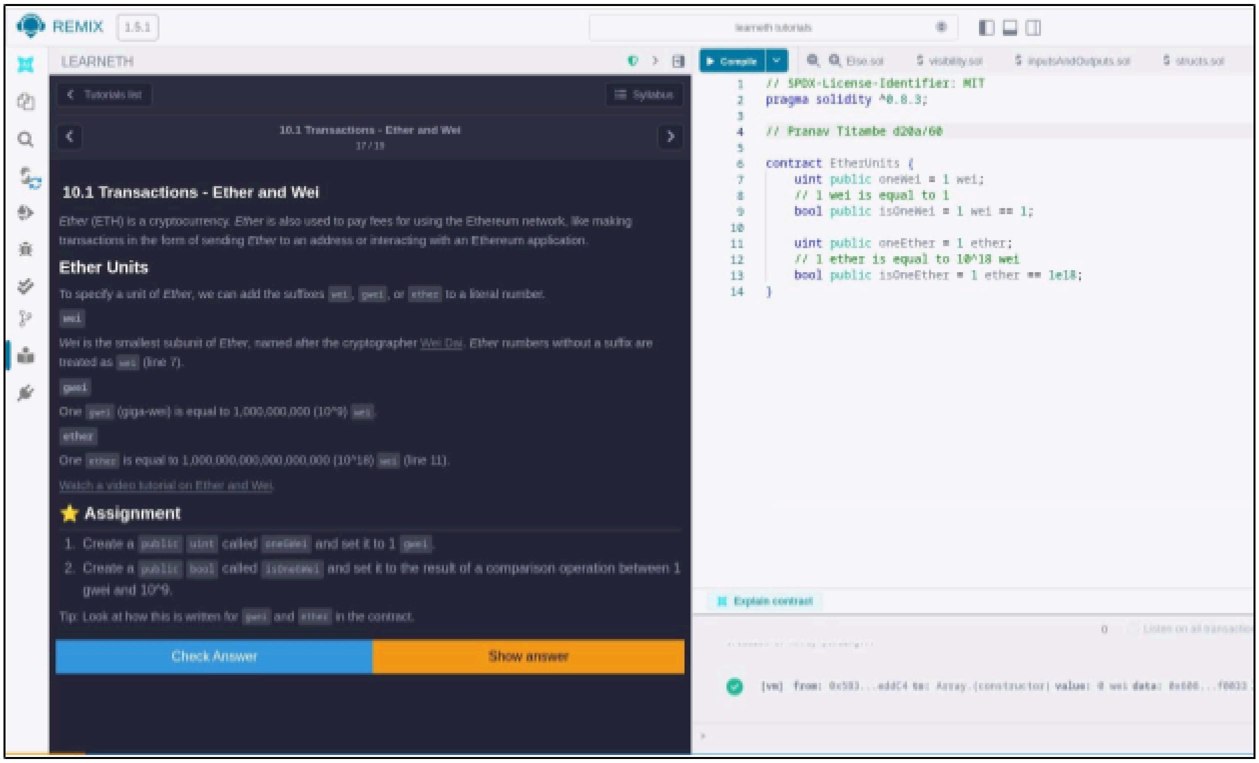
In this contract, the parameter `_arr` (line 35) has the data location calldata. If we wanted to assign a new value to the first element of the array `_arr`, we could do that in the `function g` (line 31) but not in the `function h` (line 35). This is because `_arr` in `function g` has the data location memory and *function h* has the data location `calldata`.

Assignments

```
1   // SPDX-License-Identifier: MIT
2   pragma solidity ^0.8.3;
3
4   // Pranav Titambe d20a/60
5
6   contract DataLocations {
7       uint[] public arr;
8       mapping(uint => address) map;
9       struct MyStruct {
10          uint foo;
11      }
12      mapping(uint => MyStruct) myStructs;
13
14      function f() public {    372 gas
15          // call _f with state variables
16          _f(arr, map, myStructs[1]);
17
18          // get a struct from a mapping
19          MyStruct storage myStruct = myStructs[1];
20          // create a struct in memory
21          MyStruct memory myMemStruct = MyStruct(0);
22      }
23
24      function _f(    undefined gas
25          uint[] storage _arr,
26          mapping(uint => address) storage _map,
27          MyStruct storage _myStruct
28      ) internal {
29          // do something with storage variables
30      }
31
32      // You can return memory variables
```

```
[vm] from: 0x5B3...eddC4 to: Array.(constructor) value: 0 wei data: 0x608...f0033
```

Tutorial 17



10.1 Transactions - Ether and Wei

Ether (ETH) is a cryptocurrency. *Ether* is also used to pay fees for using the Ethereum network, like making transactions in the form of sending *Ether* to an address or interacting with an Ethereum application.

Ether Units

To specify a unit of *Ether*, we can add the suffixes `wei`, `gwei`, or `ether` to a literal number.

`wei`

Wei is the smallest subunit of *Ether*, named after the cryptographer Wei Dai. *Ether* numbers without a suffix are treated as `wei` (line 7).

`gwei`

One `gwei` (giga-wei) is equal to 1,000,000,000 (10^9) `wei`.

`ether`

One `ether` is equal to 1,000,000,000,000,000,000 (10^18) `wei` (line 11).

Watch a video tutorial on Ether and Wei.

⭐ Assignment

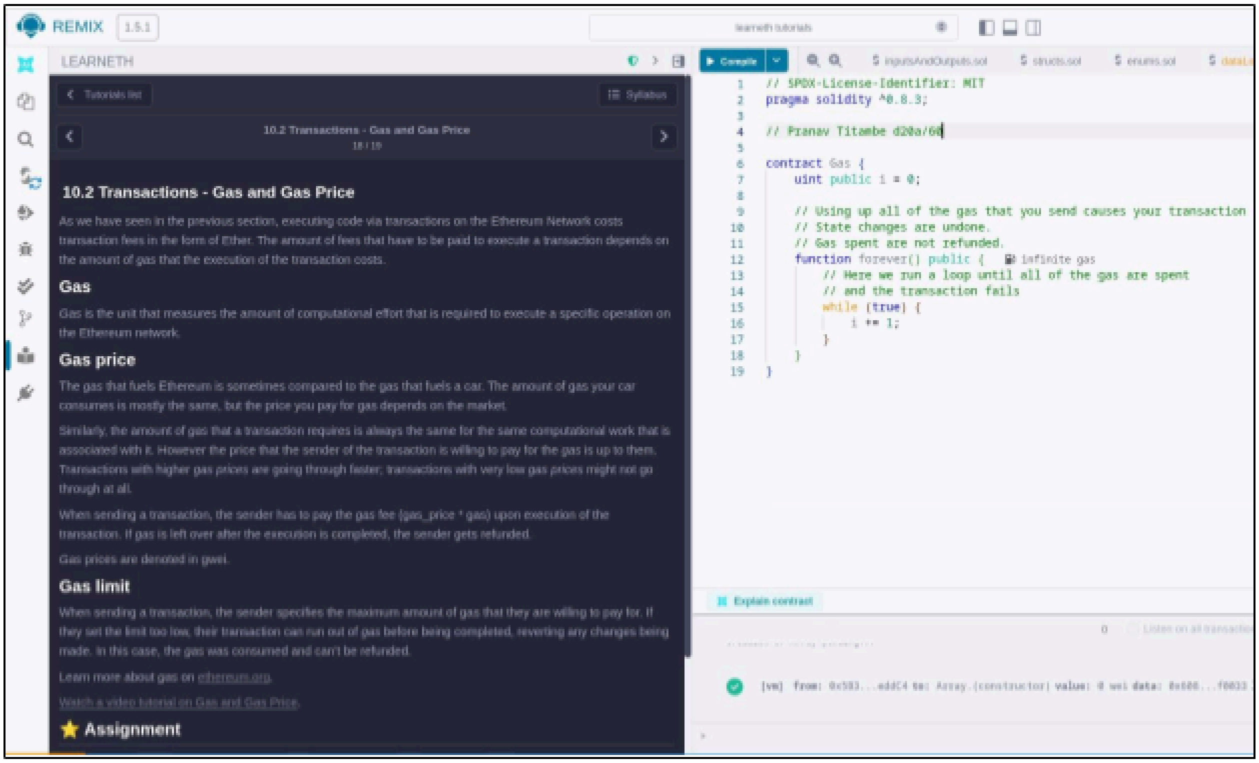
1. Create a `public` `uint` called `oneGwei` and set it to 1 `gwei`.
2. Create a `public` `bool` called `isOneGwei` and set it to the result of a comparison operation between 1 gwei and 10^9.

Tip: Look at how this is written for `gwei` and `ether` in the contract.

```
1   // SPDX-License-Identifier: MIT
2   pragma solidity ^0.8.3;
3
4   // Pranav Titambe d20a/60
5
6   contract EtherUnits {
7       uint public oneWei = 1 wei;
8       // 1 wei is equal to 1
9       bool public isOneWei = 1 wei == 1;
10
11      uint public oneEther = 1 ether;
12      // 1 ether is equal to 10^18 wei
13      bool public isOneEther = 1 ether == 1e18;
14  }
```

```
[vm] from: 0x5B3...eddC4 to: Array.(constructor) value: 0 wei data: 0x608...f0033
```

Tutorial 18



10.2 Transactions - Gas and Gas Price

As we have seen in the previous section, executing code via transactions on the Ethereum Network costs transaction fees in the form of Ether. The amount of fees that have to be paid to execute a transaction depends on the amount of gas that the execution of the transaction costs.

Gas

Gas is the unit that measures the amount of computational effort that is required to execute a specific operation on the Ethereum network.

Gas price

The gas that fuels Ethereum is sometimes compared to the gas that fuels a car. The amount of gas your car consumes is mostly the same, but the price you pay for gas depends on the market.

Similarly, the amount of gas that a transaction requires is always the same for the same computational work that is associated with it. However the price that the sender of the transaction is willing to pay for the gas is up to them. Transactions with higher gas prices are going through faster; transactions with very low gas prices might not go through at all.

When sending a transaction, the sender has to pay the gas fee (gas_price * gas) upon execution of the transaction. If gas is left over after the execution is completed, the sender gets refunded.

Gas prices are denoted in gwei.

Gas limit

When sending a transaction, the sender specifies the maximum amount of gas that they are willing to pay for. If they set the limit too low, their transaction can run out of gas before being completed, reverting any changes being made. In this case, the gas was consumed and can't be refunded.

Learn more about gas on ethereum.org.

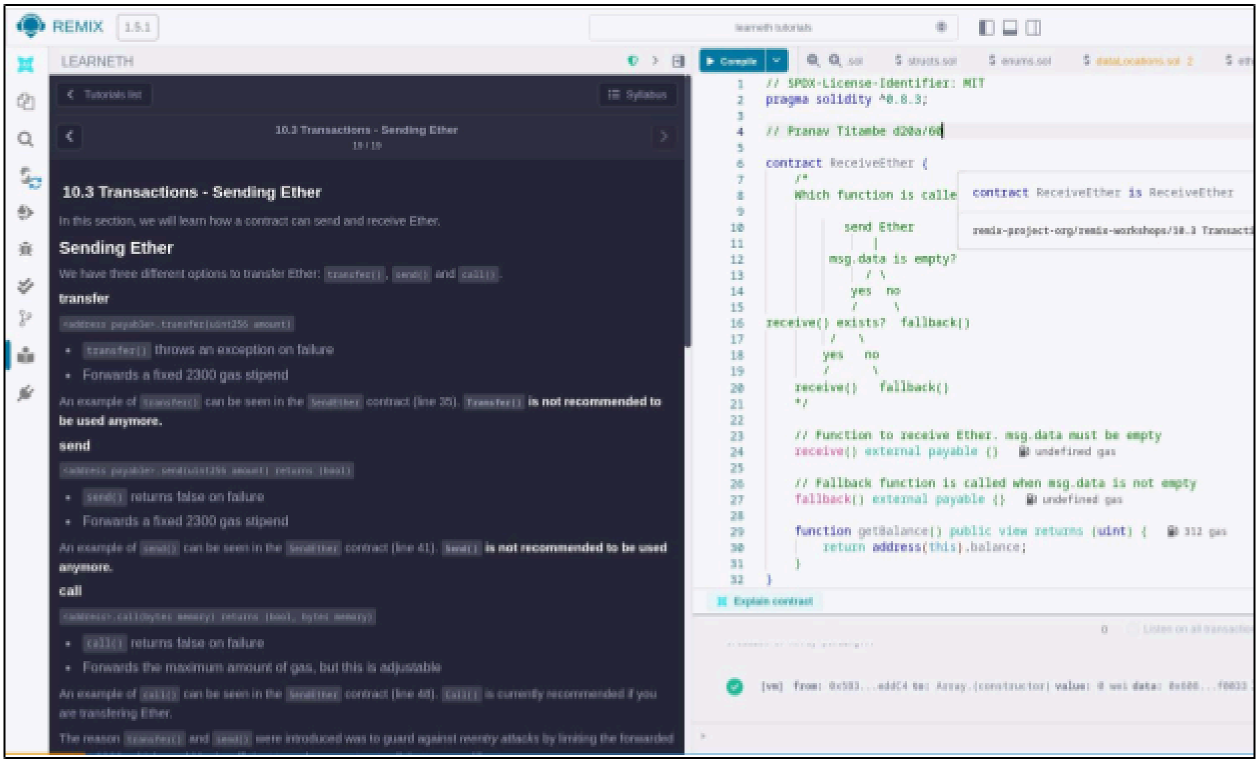
Watch a video tutorial on Gas and Gas Price.

⭐ Assignment

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.3;

// Pranav Titambe d20a/60

contract Gas {
    uint public i = 0;

    // Using up all of the gas that you send causes your transaction
    // State changes are undone.
    // Gas spent are not refunded.
    function forever() public {    infinite gas
        // Here we run a loop until all of the gas are spent
        // and the transaction fails
        while (true) {
            i += 1;
        }
    }
}
```

```
[vm] from: 0x5B3...eddC4 to: Array.(constructor) value: 0 wei data: 0x608...f0033
```

Tutorial 19



10.3 Transactions - Sending Ether
19 / 19

10.3 Transactions - Sending Ether

In this section, we will learn how a contract can send and receive Ether.

Sending Ether

We have three different options to transfer Ether: `transfer()`, `send()` and `call()`.

transfer

`<address payable>.transfer(uint256 amount)`

- `transfer()` throws an exception on failure
- Forwards a fixed 2300 gas stipend

An example of `transfer()` can be seen in the `SendEther` contract (line 35). `Transfer()` **is not recommended to be used anymore.**

send

`<address payable>.send(uint256 amount) returns (bool)`

- `send()` returns false on failure
- Forwards a fixed 2300 gas stipend

An example of `send()` can be seen in the `SendEther` contract (line 41). `Send()` **is not recommended to be used anymore.**

call

`<address>.call(bytes memory) returns (bool, bytes memory)`

- `call()` returns false on failure
- Forwards the maximum amount of gas, but this is adjustable

An example of `call()` can be seen in the `SendEther` contract (line 48). `Call()` is currently recommended if you are transfering Ether.

The reason `transfer()` and `send()` were introduced was to guard against reentry attacks by limiting the forwarded

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.3;

// Pranav Titambe d20a/60

contract ReceiveEther {
    /*
    Which function is calle

            send Ether
                |
          msg.data is empty?
               / \
             yes  no
             /      \
receive() exists?  fallback()
         /   \
        yes   no
        /      \
    receive()   fallback()
    */

    // Function to receive Ether. msg.data must be empty
    receive() external payable {}    undefined gas

    // Fallback function is called when msg.data is not empty
    fallback() external payable {}    undefined gas

    function getBalance() public view returns (uint) {    312 gas
        return address(this).balance;
    }
}
```

```
contract ReceiveEther is ReceiveEther
remix-project-org/remix-workshops/10.3 Transacti
```

```
[vm] from: 0x5B3...eddC4 to: Array.(constructor) value: 0 wei data: 0x608...f0033
```

Conclusion:

This experiment provided an in-depth exploration of Solidity programming through structured practical implementation using the Remix IDE. Fundamental concepts—including data types, variable classifications, function definitions, visibility specifiers, modifiers, constructors, control flow mechanisms, data structures, and transaction management—were systematically implemented and evaluated.

The process of designing, compiling, and deploying smart contracts on the Remix Virtual Machine (VM) enabled a comprehensive understanding of smart contract architecture and blockchain execution mechanisms. The practical exposure strengthened conceptual clarity and technical proficiency in Ethereum-based development.